

November 2005

# Scalable Store-Load Forwarding via Store Queue Index Prediction

Tingting Sha
*University of Pennsylvania*

Milo Martin
*University of Pennsylvania*, milom@cis.upenn.edu

Amir Roth
*University of Pennsylvania*, amir@cis.upenn.edu

# Scalable Store-Load Forwarding via Store Queue Index Prediction

**Abstract**
Conventional processors use a fully-associative store queue (SQ) to implement store-load forwarding. Associative search latency does not scale well to capacities and bandwidths required by wide-issue, large window processors. In this work, we improve SQ scalability by implementing store-load forwarding using speculative indexed access rather than associative search. Our design uses prediction to identify the single SQ entry from which each dynamic load is most likely to forward. When a load executes, it either obtains its value from the predicted SQ entry (if the address of the entry matches the load address) or the data cache (otherwise). A forwarding mis-prediction — detected by pre-commit filtered load re-execution — results in a pipeline flush. SQ index prediction is generally accurate, but for some loads it cannot reliably identify a single SQ entry. To avoid flushes on these difficult loads while keeping the single-SQ-access-per-load invariant, a second predictor delays difficult loads until all but the youngest of their "candidate" stores have committed. Our predictors are inspired by store-load dependence predictors for load scheduling (Store Sets and the Exclusive Collision Predictor) and unify load scheduling and forwarding.

Experiments on the SPEC2000 and MediaBench benchmarks show that on an 8-way issue processor with a 512-entry reorder buffer, our technique performs within 3.3% of an ideal associative SQ (same latency as the data cache) and either matches or exceeds the performance of a realistic associative SQ (slower than data cache) on 31 of 47 programs.

# Scalable Store-Load Forwarding via Store Queue Index Prediction

Tingting Sha, Milo M.K. Martin, Amir Roth
Department of Computer and Information Science, University of Pennsylvania
{shatingt,milom,amir}@cis.upenn.edu

**Abstract**

*Conventional processors use a fully-associative store queue (SQ) to implement store-load forwarding. Associative search latency does not scale well to capacities and bandwidths required by wide-issue, large window processors. In this work, we improve SQ scalability by implementing store-load forwarding using speculative indexed access rather than associative search. Our design uses prediction to identify the single SQ entry from which each dynamic load is most likely to forward. When a load executes, it either obtains its value from the predicted SQ entry (if the address of the entry matches the load address) or the data cache (otherwise). A forwarding mis-prediction—detected by pre-commit filtered load re-execution—results in a pipeline flush. SQ index prediction is generally accurate, but for some loads it cannot reliably identify a single SQ entry. To avoid flushes on these difficult loads while keeping the single-SQ-access-per-load invariant, a second predictor delays difficult loads until all but the youngest of their "candidate" stores have committed. Our predictors are inspired by store-load dependence predictors for load scheduling (Store Sets and the Exclusive Collision Predictor) and unify load scheduling and forwarding.*

*Experiments on the SPEC2000 and MediaBench benchmarks show that on an 8-way issue processor with a 512-entry reorder buffer, our technique performs within 3.3% of an ideal associative SQ (same latency as the data cache) and either matches or exceeds the performance of a realistic associative SQ (slower than data cache) on 31 of 47 programs.*

## 1. Introduction

Store-load forwarding is a critical aspect of dynamically scheduled execution. Conventional processors implement store-load forwarding by buffering the addresses and data values of all in-flight stores in an ***age-ordered store queue (SQ)***. A load accesses the data cache and in parallel associatively searches the SQ for older stores with matching addresses. The load obtains its value from the youngest such store (if any) or from the data cache, as illustrated in Figure 1(a).

Associative structures can be made fast, but often at the cost of substantial additional energy, area, and/or design effort. Furthermore, these implementation disadvantages compound super-linearly—especially for *ordered* associative structures like the SQ—as structure size or bandwidth scales up. As SQ access is on the load execution critical path, fully-associative search of a large SQ can result in load latency that is longer than data cache access latency, which in turn complicates scheduling and introduces replay overheads [8].

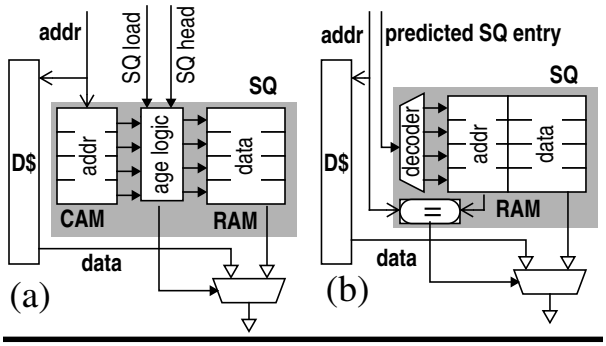


**Figure 1. Store queues:** (a) associative, (b) indexed.

We introduce a scalable SQ design that implements store-load forwarding without associative search. As each dynamic load is renamed, we use store-load dependence prediction [3, 9, 22] to predict the single in-flight store from which that load is most likely to forward. As illustrated in Figure 1(b), when a load executes, it accesses the SQ only at this predicted index, not associatively. If the entry contains a matching address, the load reads the corresponding data value. Otherwise, it uses the value from the cache. Because indexed forwarding is speculative, we use filtered in-order load re-execution prior to commit [2, 16] to catch mis-predictions (which trigger pipeline flushes) and train the store-load dependence predictor.

To predict forwarding SQ entries, we use a two-table predictor that is an adaptation of Store Sets [3]. The first table maps each dynamic load to a small set of static stores from which it has forwarded in the past; the second table maps each of these static stores (PCs) to the SQ index of its youngest in-flight instance. The predictor selects the youngest of these indices. Our experiments show that this predictor mis-forwards (i.e., misses an actual forwarding and incurs a flush) only 0.18% of dynamic loads (less than 2 in 1000).

For a few loads—especially in large windows which support more (and more complex) forwarding patterns—the forwarding predictor cannot reliably choose a single candidate forwarding store. To minimize flushing while maintaining the single-SQ-access-per-load simplification, we delay difficult loads until all but the youngest of their candidate forwarding stores have committed. We use a distance-based dependence predictor, similar to the Exclusive Collision predictor [22], to map each static load to a maximum number of older stores that can safely be in-flight for the load to forward correctly. This delay mechanism reduces mis-forwarding to 0.03% at the cost of delaying the execution of 2.3% of loads by an average of 53 cycles each.

Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)
0-7695-2440-0/05 $20.00 © 2005 IEEE

We are not the first researchers to propose alternative SQ designs to scale store-load forwarding to large window sizes and wider issue [1, 5, 13, 15, 17, 20]. Our approach differs from these previous proposals as it completely eliminates associative search while maintaining the simplicity of a non-segmented, age-ordered SQ organization. Our design unifies load scheduling and store-load forwarding in a single mechanism and transfers the complexity of that mechanism from the latency critical execution core to the more latency tolerant front end. When combined with recent proposals for non-associative load queues [2, 16], our design yields an in-flight data memory system that is completely free of associative search and eliminates one of the structural barriers to wide-issue large-window processors.

## 2. Baseline Microarchitecture: Background

This section describes four aspects of out-of-order load and store execution in modern processors: (1) committing stores to the data cache in program order, (2) forwarding values to loads from the youngest older in-flight stores that wrote to the address, (3) detecting memory-ordering violations by determining when a load executed too early relative to its producing store, and (4) reducing the frequency of memory-ordering violations. The first two functions are performed by a store queue (SQ), the third by a load queue (LQ), and the fourth by a load scheduling predictor. Although this paper focuses on the SQ, our design uses and dovetails with previously proposed scheduling and ordering techniques. This section reviews these techniques, focusing on those we incorporate into our microarchitecture.

**Store commit and store-load forwarding.** Conventional processors implement in-order store commit and store load forwarding with an age-ordered SQ**,** an array that contains one entry for each in-flight store in program (age) order. Each SQ entry encodes the store's physical address, data size, ready bits, and value. The SQ supports three operations: indexed writes for store execution, indexed reads for store commit, and fully-associative search-and-read operations for load execution. If a load forwards (i.e., receives its value from an in-flight store via the SQ), it must do so from the youngest in-flight store older than itself that has a matching address. To quickly find all matching store addresses (there may be several), the address portion of the SQ is implemented as a CAM (content addressable memory). A priority encoding age logic follows the CAM and selects the youngest matching store that is also older than the load. The associative search logic (the CAM and priority encoder) is the slow and non-scalable component of the SQ, and is the one our design eliminates.

**Detecting memory-ordering violations.** A memory-ordering violation occurs when a load executed too early (i.e., before the store upon which the load depends executed). Modern processors detect memory-ordering violations using a load queue (LQ). Similar to an SQ, a traditional LQ is a CAM that contains load addresses in program order. When a load executes, it writes its address into the LQ. When a store executes, it associatively searches the LQ for younger loads that read the address it wrote. A match indicates an ordering violation and triggers a pipeline flush.

To avoid expensive associative search, memory ordering violations can alternatively be detected by in-order load re-execution prior to commit [2, 6]. This approach detects a violation when a load's re-executed value does not equal its (initial) executed value. To reduce data cache traffic, only those loads that execute in the presence of older stores with unknown addresses are re-executed. For SPECint, this is about 9% of loads.

Store Vulnerability Window (SVW) [16] further reduces the re-execution rate. With SVW, a load re-executes only if it issued in the presence an older store with an unknown address *and* that store wrote to the load's address. SVW assigns each store a monotonically increasing sequence number (the Store Sequence Number or SSN). An address-indexed table called the Store Sequence Bloom Filter (SSBF) tracks the SSN of the most recent committed store to a given address. When a load executes, the SSN of the youngest older store to which it is not vulnerable—the SSN of the forwarding store or the SSN of the youngest committed store—is recorded in its LQ entry. Prior to re-execution, the load uses its address to probe the SSBF. It re-executes only if the SSN in the SSBF entry is greater than the SSN in its own LQ entry, i.e., if its address collides with that of a store to which it is vulnerable. With SVW, the SPECint re-execution rate falls to 1%.

**Reducing memory-ordering violations.** Modern processors reduce memory-ordering violations by recording the identities of offending loads and delaying the execution of future instances of those loads enough to avoid violations (and ideally without introducing unnecessary delay). Simple store-blind predictors [7] delay suspect loads until *all* older stores execute. More sophisticated store-load pair predictors [3, 9, 22] force the load to wait for a particular store to execute.

In a processor with a traditional LQ, which detects memory-ordering violations during store execution, the PCs of both the store and the load involved in the violation are readily available to train a store-load pair predictor. However, pre-commit re-execution does not automatically identify which store caused the violation [2]. This limitation is overcome by the Store PC Table (SPCT), a small, address-indexed table (similar to the SSBF) that holds the PC of the last committed store to write to each address [16].

**Baseline microarchitecture.** Our baseline microarchitecture uses SVW-filtered load re-execution [16] and a load scheduling predictor inspired by Store Sets [3]. This sophisticated predictor allows our baseline to use address-less scheduling [11] and avoid splitting stores into address and data operations. Figure 2 shows the load-store unit of this microarchitecture. Notice the absence of the LQ address CAM and the modifications to support SVW-filtered re-execution in gray.

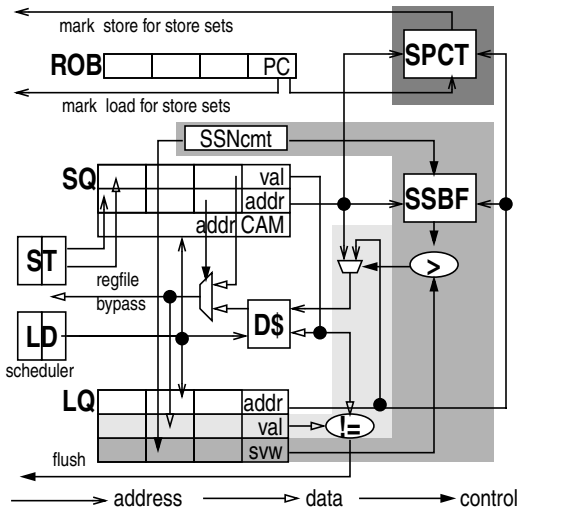


**Figure 2. Baseline load-store unit.** This design enforces memory ordering using SVW-filtered re-execution (note the absence of an LQ address CAM) using three sets of structures. The load re-execution port and LQ value fields (light gray) implement basic re-execution. The LQ SVW fields and SSBF (medium gray) implement the SVW filter. The SPCT (dark gray) helps train store-load pair predictors.

## 3. Indexed Store-Load Forwarding

Our design improves SQ scalability by replacing fully-associative search traditionally used to implement store-load forwarding with indexed (i.e., direct) SQ access. As illustrated in Figure 1(b), this approach reduces the load-forwarding critical path by replacing the address CAM and age logic with a simple decoder.

Indexed SQ access uses two predictors. To avoid search, we use a ***forwarding index predictor*** to predict the SQ index from which a load is most likely to forward. No forwarding predictor is perfectly accurate. To reduce the frequency of forwarding mis-predictions, we use a ***delay index predictor*** to delay the execution of troublesome loads until all non-predicted stores likely to forward to this load have committed, allowing the load to obtain the correct value from either the cache or the single predicted SQ entry. Before describing our specific design of these two predictors (in Section 3.2 and Section 3.3, respectively), we first describe the mechanism that uses the predictions to perform store-load forwarding and load scheduling.

### 3.1. Indexed Forwarding Mechanism

The processor generates a forwarding index prediction and a delay index prediction for each dynamic load during the decode/rename pipeline stages. Both predictions either identify a particular in-flight store instance or predict that no in-flight store instance is relevant. The load executes when (1) its input registers are ready, (2) the store corresponding to its forwarding index has *executed* (to enforce the likely memory dependence), and (3) the store corresponding to its delay index has *committed* (to force ambiguous but potentially value-producing stores to commit to the cache). Notice that in addition to coordinating forwarding, use of these two predictions subsume traditional load scheduling. During load execution, the load accesses the data cache in parallel with reading the single SQ entry determined by the predicted forwarding index. If (1) the address of the predicted SQ entry matches the load address and (2) the load width is less than or equal to the store width, the load uses the data value from the SQ. Otherwise, the load receives its value from the cache. We use SVW-filtered load re-execution (described in Section 2) to detect forwarding mis-predictions (which trigger pipeline flushes) and train both predictors.

**SQ indices and SSNs.** Indexed SQ access requires an SQ index (position). However, SQ indices do not contain information about committed stores and require somewhat complex wrap-around logic. To simplify several prediction, scheduling, and re-execution functions and to uniquely identify recent store instances (both in-flight *and* committed), we name stores using their Store Sequence Numbers (SSNs) as defined by SVW [16]. A given store is in-flight if its SSN is greater than the global counter $SSN_{cmt}$. The SQ index of an in-flight store is derived from the low-order bits of its SSN (assuming SQ size is a power of two). As such, we refer to the forwarding index as $SSN_{fwd}$ and the delay index as $SSN_{dly}$. Wrap-around of $N$-bit SSNs is handled by draining the pipeline and clearing all structures that hold SSNs when a store with SSN=0 is renamed (once every $2^N$ stores).

### 3.2. Forwarding Index Predictor

The goal of a forwarding index predictor is to determine the store instance (identified by its SSN) that is most likely to forward to a given load instance. In our scheme, forwarding takes place only if the load's address matches that of the predicted store. Only missed forwarding instances (i.e., loads that should obtain values from an in-flight store that we fail to predict) are considered incorrect. Predicted forwardings that result in the load properly obtaining its value from the cache are still considered correct. We call the first case a ***mis-forwarding*** and the second a ***non-forwarding***. By tolerating non-forwarding, we relax the precision requirements of the forwarding predictor.

Our forwarding predictor consists of two tables. The ***Forwarding Store Predictor (FSP)*** maps each load PC to a small set of store PCs from which the load recently forwarded. The FSP is a PC-indexed, set-associative table. Each entry contains a valid bit, a partial tag, a partial store PC, and a short saturating counter. The associativity of the FSP determines both how many loads can share a set and how many store dependences a single load can represent. Our experiments show that 2-way set-associativity is adequate. The ***Store Alias Table (SAT)*** maps each store PC to the SSN of the youngest in-flight instance of that store. The SAT is untagged and each entry contains only a single SSN.

Chained FSP/SAT access is used to attach to each dynamic load the SSN of the most likely-to-forward

dynamic store (this is the load's $SSN_{fwd}$). The decode stage uses the load PC to access the FSP and produce a small set of store PCs (limited by FSP associativity). The rename stage accesses the SAT (in parallel for each of the store PCs returned by the FSP) to generate a set of SSNs. The youngest (largest) of these SSNs is chosen as the load's predicted $SSN_{fwd}$.

**SAT update.** The SSN of each store is inserted into the SAT at rename. Like a register alias table (RAT), the SAT is repaired on pipeline flushes, although a SAT requires repair only for performance, not for correctness. The mechanisms for repairing the SAT—logging over-written entries or checkpointing—are analogs of the mechanisms that repair a RAT.

**FSP training**. The FSP is trained at load commit with the help of the SPCT and SSBF [16]. The SPCT maps each (partial) address to the PC of the last store to write to the address, allowing each committing load to determine the PC of the store it should have forwarded from, if any. The SSBF maps each (partial) address to the SSN of the last store to write to it, allowing the load to determine the distance (in dynamic stores) to this forwarding store. Distance information is useful because a distance greater than the size of the SQ means that no forwarding could have actually occurred. Both the SSBF and SPCT are implemented at a granularity of 1 byte, with wide stores making multiple writes and wide loads making multiple reads. This organization is needed to capture forwarding of multiple data sizes, and can be implemented efficiently by banking each structure 8-ways (assuming a maximum data size of 8 bytes).

If making predictions for non-forwarding loads had no negative effects, the FSP could be trained (up) only by mis-forwarding loads. However, a load must wait until its predicted forwarding store has executed, even in non-forwarding cases. If a load forwards from a store 1 out of 1000 times, it is better not to learn (or to unlearn) the forwarding behavior and incur a single flush rather than to unnecessarily delay the other 999 instances. For this reason, the FSP is potentially trained (either positively or negatively) by every committing load. The counter in each entry weighs positive training against negative (our default ratio is 8:1).

Generally speaking, we learn store-load dependences on correct forwarding (to reinforce dependences known to be useful) and on mis-forwardings in which we fail to predict not only the forwarding index, but also the forwarding store PC (to create new, potentially-useful dependences). We unlearn dependences when the load and most recent store to its address are dynamically far enough apart (i.e., further than SQ size) that no actual forwarding can take place (to unlearn entries that result in too many non-forwardings). We also unlearn dependences if we successfully predict the forwarding store's PC, but not its dynamic instance. This happens when a load forwards from what is not the most recent instance of a store (e.g., a load forwards across multiple loop iterations as in the loop body `X[i]=A*X[i-2]`). Because our SAT tracks only the most recent instance of each store, our mechanism cannot capture this ***not-most-recent forwarding*** behavior. All the same, there is no point in delaying the load on a store instance on which it is known not to depend (e.g., load `X[5]` depends on store `X[3]`, but there is no point in delaying it on store `X[4]` even though that is the only store the SAT can predict). Section 3.3 describes how the delay predictor prevents pipeline flushes for loads that exhibit not-most-recent forwarding behavior.

**Example operation.** Figure 3 shows our indexed SQ in operation for a 2-entry FSP, a 2-entry SAT, and a 4-entry SQ. For simplicity of the example, each FSP entry has a single store (i.e., the FSP is direct-mapped). The main participants are dynamic instances of static stores Y and Z and static load W. The left hand side of the figure shows a predictor training sequence involving one execution of these three instructions; the right hand side shows a successful indexed forwarding that uses the dependence information learned during the first execution. Each sequence consists of five snapshots which show the relevant events in the lives of the participants. Each snapshot shows the contents of five structures: (1) the FSP, (2) the SAT, (3) an SQ which is marked by head and tail pointers (thick lines) and in which each entry contains a store PC (Y or Z), a data address (A or B), a value (single digit), and an implicit SSN (double digit number above the entry), (4) the data cache (D$), and (5) the SPCT, both of which are indexed by data address (A or B). We first examine the training sequence:

1. **Store Z renames;** enqueues on the SQ, receives the next sequential SSN (18) and notes in the SAT that the youngest instance of store Z has SSN 18. **Load W decodes** and accesses the FSP, but finds no forwarding store.
2. **Store Z (SSN 18) executes** and writes its address/value (B/6) to the SQ. **Load W renames** but has no store PC with which to access the SAT.
3. **Store Y (older than Z) commits** and writes its value (5) and PC (Y) to the D$ and SPCT, respectively, in the slots corresponding to its address (A). **Load W calculates its address (B) and executes.** Because it was not predicted to forward, it reads the value (0) from the D$. This is a mis-forwarding; it should actually read its value from store Z (SSN 18).
4. **Store Z commits** and writes its value (6) and PC (Z) to the D$ and SPCT, respectively, in the slots corresponding to its address (B).
5. **Load W re-executes** and discovers a discrepancy between the value it originally loaded (0) and the correct value it re-loaded (6). The load triggers a flush, learns the identity of the store it should have forwarded from (Z) by accessing the SPCT using its address (B), and enters that store into its FSP entry.

The sequence on the right of the figure follows the same events in a future execution of these three instructions. The difference in this sequence is that the forwarding relationship between store Z and load W has been previously established in the FSP.

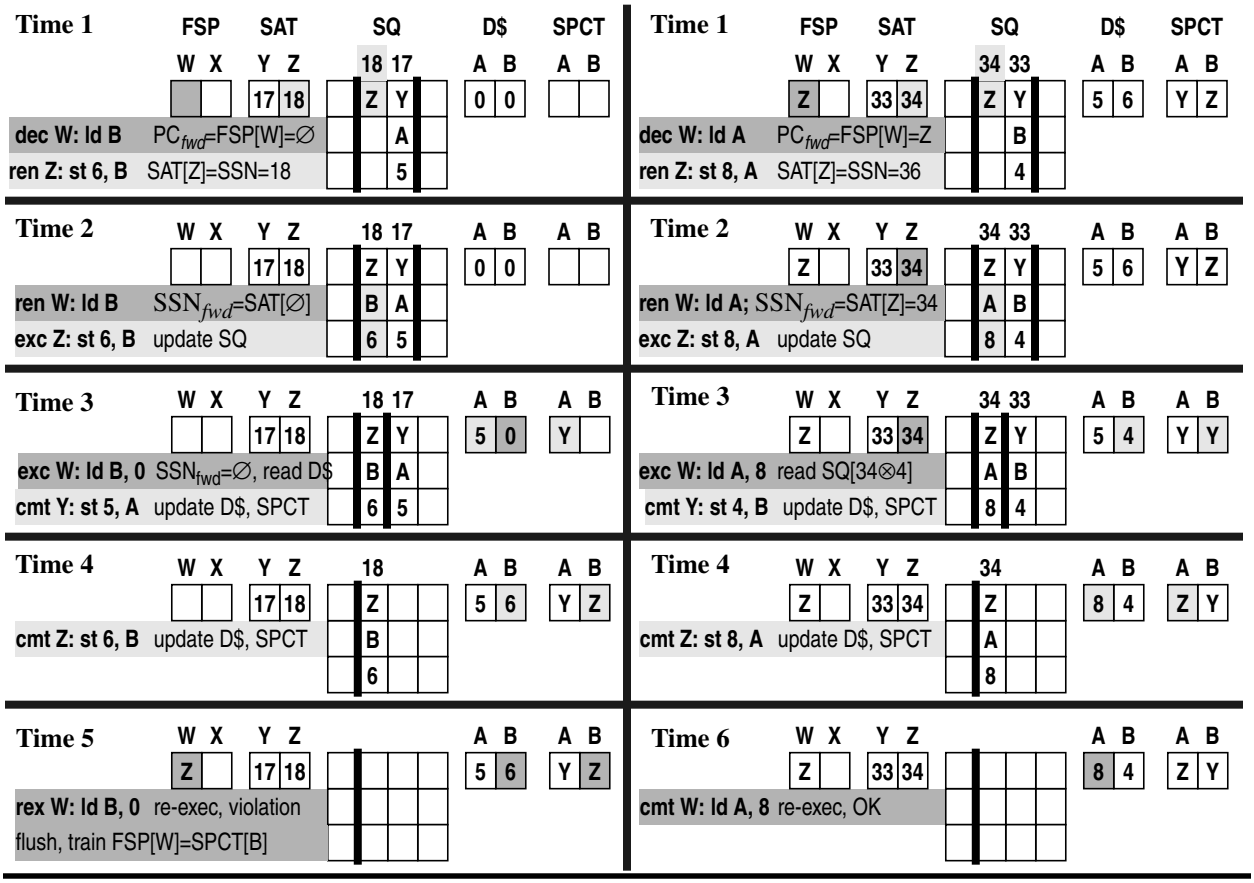


**Figure 3. Working example. LEFT:** forwarding predictor training sequence. **RIGHT:** speculative forwarding sequence.

1. **Store Z renames,** enqueues on the SQ, is assigned the next SSN (34) and notes it in its SAT entry. **Load W decodes,** accesses the FSP, and this time finds that it may forward from store Z.
2. **Store Z executes** and writes its address/value (A/8) to the SQ. **Load W renames**, accesses the SAT using its predicted forwarding store PC (Z) and finds that its likely forwarding SQ entry corresponds to SSN 34 ($SSN_{fwd}$ = 34).
3. **Store Y commits** and updates the D$, SPCT, and SAT. **Load W calculates its address (A) and executes.** It indexes the SQ at index (34 mod 4) and finds a matching address (A). It therefore reads the value from the same SQ entry (8).
4. **Store Z commits**, updating the D$ (8) and SPCT (Z) at its address (A).
5. **Load W re-executes**, discovers that the value it originally forwarded (8) is correct, and commits.

**Similarity to Store Sets.** The organization of our predictor is similar to (and inspired by) a Store Sets predictor, which predicts store-load pair dependences for load-scheduling purposes [3]. Our FSP is the analog of Store Set ID Table (SSIT), but whereas the FSP maps load PCs to store PCs directly, the SSIT maps both load and store PCs to Store Set IDs. Our SAT is the analog of the Last Fetched Store Table (LFST).

### 3.3. Delay Index Predictor

The goal of delay index prediction is to reduce mis-forwarding flushes by delaying execution of difficult-to-forward-predict loads. For each difficult load, we predict a delay index ($SSN_{dly}$); the load does not execute until the corresponding store commits. An ideal predictor delays as few loads as possible for as few cycles as possible to avoid mis-forwarding.

Our delay predictor consists of one table. The ***Delay Distance Predictor (DDP)*** maps each static load to the distance (in dynamic stores) between the load and the closest older store that causes its mis-forwardings. The DDP is a tagged, PC-indexed table. Each entry has a valid bit, a partial tag, a saturating counter. and two distance fields. The counter determines *if* a load should be delayed. The distance fields are represented using $\lceil \log_2(SQ.size) \rceil$ bits because any delay distance larger than the size of the SQ is effectively no delay at all. As described below, the second distance field facilitates delay distance down-training.

At decode, each load accesses the DDP to obtain a delay distance $D_{dly}$. At rename, $SSN_{dly}$ is computed as the SSN of the most recently renamed store ($SSN_{ren}$) minus the distance field ($D_{dly}$). If the load has no DDP entry or the entry's counter is below threshold, the predicted $SSN_{dly}$ is 0, indicating no effective delay.

**DDP training.** Like the FSP, the DDP is trained by all committing loads and supports both positive and negative training. Generally speaking, we learn delay on a wrong forwarding prediction and unlearn it on correct forwardings (if we can correctly predict the forwarding behavior of a load, there is no need to delay it until the forwarding store commits). Note, a wrong forwarding prediction does not necessarily mean a mis-forwarding. A load with an incorrect forwarding prediction can still obtain its correct value from the cache. This happens when the actual forwarding store has already committed, either naturally or via a forced delay.

On any wrong forwarding prediction, the DDP increments the delay counter and learns a delay distance equal to the difference of $SSN_{cmt}$ and the SSN of the actual forwarding store (retrieved from the SSBF). To conservatively preserve information about previous delays, a delay distance is learned only if it is smaller than the current known delay. On a correct forwarding prediction, the DDP decrements the delay counter. To allow unlearning of delay distances (in addition to binary delay-or-not decisions), the predictor entry uses a second "future" distance field. Both distance fields are trained in parallel. Every 8 load instances the "current" field is set to the future field and the future field is reset. This mechanism allows loads to avoids monotonic convergence to overly conservative delay distances.

The cooperation between forwarding and delay index prediction is illustrated by the example that forwarding prediction itself cannot handle, not-most-recent forwarding (e.g., as in the loop `X[i]=A*X[i-2]`). We have already seen that the forwarding predictor will not learn to forward load `X[5]`. However, because the forwarding prediction will always be wrong, the delay predictor will properly learn to delay the load until store `X[3]` commits. This example also motivates why we use distances (rather than the SAT) to compute delays. The SAT can identify only the most recent instance of each store; a distance can identify any store instance.

**Similarity to Exclusive Collision Predictor.** Our delay distance predictor is similar to the Exclusive Collision predictor [22] and both are used for load scheduling. The Exclusive Collision predictor was used with an associative SQ to delay all loads until some (potentially empty) range of older stores has *executed*. Our predictor is used with an indexed SQ to delay *only difficult loads* until some range of older stores has *committed*.

## 3.4. Summary

Table 1 summarizes actions for loads and stores at each pipeline stage for three SQ configurations. The first uses an associative SQ, Store Sets scheduling, and SVW-filtered load re-execution (whose actions are in bold). This configuration represents research proposals that preceded this paper.

The second configuration is our baseline. It uses an associative SQ, but a Store Sets scheduler reformulated using PCs/SSNs rather than SSIDs/INUMs (equivalent of SQ indices). The important differences between the original Store Sets and our formulation are: (1) Store Sets can represent a arbitrary number of store dependences per load whereas we are limited by FSP associativity; (2) Store Sets serializes the execution of all dynamic loads and stores within a set whereas we only serialize a load with a single dynamic store.

The final configuration is our proposed speculative indexed SQ. The modifications over a configuration that uses re-execution and reformulated Store Sets are limited. The notable differences are the indexed SQ access at execute (of course) and the delay machinery.

**Table 1. Pipeline action diagram.** Store-load forwarding relevant actions (for both loads and stores) for three store queue designs. ld.A and st.A refer to the addresses of the load and store, respectively.

| DECODE | RENAME | WAIT UNTIL | EXECUTE | SVW / RE-EXECUTE / COMMIT |
|---|---|---|---|---|
| Associative store queue with original Store Sets scheduling and **SVW-filtered load re-execution** | | | | |
| ld.SSID=SSIT[ld.PC] | ld.INUM=LFST[ld.SSID] | SQ[ld.INUM] issue | search SQ[ld.A]<br>**ld.SVW = forward?**<br>**st.SSN : $SSN_{cmt}$** | **SSBF[ld.A] > ld.SVW ? re-execute**<br>**re-execute, violation? flush**<br>**SSIT[ld.PC, SPCT[ld.A]]=ld.SSID** |
| st.SSID=SSIT[st.PC] | LFST[st.SSID]=INUM++ | | | **SSBF[st.A]=$SSN_{cmt}$++, SPCT[st.A]=st.PC** |
| Associative store queue with **reformulated Store Sets scheduling** and SVW-filtered load re-execution | | | | |
| **ld.PC$_{fwd}$=FSP[ld.PC]** | **ld.SSN$_{fwd}$=SAT[ld.PC$_{fwd}$]** | **SQ[ld.SSN$_{fwd}$] issue** | search SQ[ld.A]<br>ld.SVW = forward?<br>st.SSN : $SSN_{cmt}$ | SSBF[ld.A] > ld.SVW ? re-execute<br>re-execute, violation? flush, **recover SAT**<br>**FSP[ld.PC]=SPCT[ld.A]** |
| | **SAT[st.PC]=$SSN_{ren}$++** | | | SSBF[st.A]=$SSN_{cmt}$++, SPCT[st.A]=st.PC |
| **Indexed store queue** with reformulated Store Sets scheduling and SVW-filtered load re-execution | | | | |
| ld.PC$_{fwd}$=FSP[ld.PC]<br>**ld.D$_{dly}$=DDP[ld.PC]** | ld.SSN$_{fwd}$=SAT[ld.PC$_{fwd}$]<br>**ld.SSN$_{dly}$=SSN$_{ren}$–ld.D$_{dly}$** | SQ[ld.SSN$_{fwd}$] issue<br>**ld.SSN$_{dly}$ ≤ SSN$_{cmt}$** | **index SQ[ld.SSN$_{fwd}$]**<br>ld.SVW = forward?<br>st.SSN : $SSN_{cmt}$ | SSBF[ld.A] > ld.SVW ? re-execute<br>re-execute, violation? flush, recover SAT<br>**ld.PC$_{fwd}$ != SPCT[ld.A]**<br>**FSP[ld.PC]=SPCT[ld.A]**<br>**ld.SSN$_{fwd}$ != SSBF[ld.A] ?**<br>**DDP[ld.PC]min=SSN$_{cmt}$–SSBF[ld.A]** |
| | SAT[st.PC]=$SSN_{ren}$++ | | | SSBF[st.A]=$SSN_{cmt}$++, SPCT[st.A]=st.PC |

## 4. Experimental Evaluation

An ideal SQ has access bandwidth and latency equal to those of the data cache. Our speculative indexed SQ will never outperform an ideal associative SQ in terms of IPC. Our goal is to show our indexed SQ performs nearly as well as an ideal associative SQ—and competitively with a realistic (slow) associative SQ—while maintaining the implementation advantages we described earlier.

### 4.1. Methodology

We evaluate the indexed SQ using timing simulation on the SPEC2000 (our simulator cannot properly execute *fma3d*) and MediaBench programs. We run the SPEC programs on their training inputs using 2% periodic sampling with 8% cache/branch predictor warm-up. Each sample contains 10M instructions. We run the MediaBench programs unsampled on their provided inputs. All programs execute to completion.

**General processor configuration.** Our simulator executes the Alpha AXP user-level ISA. We model a dynamically scheduled processor with a 512-entry reorder buffer, 300-entry issue queue, 128-entry load queue, and 64-entry store queue. The pipeline has 19 stages (3 fetch, 2 decode, 2 rename, 2 schedule, 3 register read, 1 execute, 1 writeback, 1 SVW, 3 re-execute, and 1 commit). Our processor can fetch up to 12 instructions per cycle, past a single taken branch. It predicts branches using a 4K-entry hybrid gShare/bimodal predictor, a 2K-entry, 4-way set-associative BTB, and a 32-entry RAS. Our processor can decode, rename, issue, and commit 8 instructions per cycle. The issue mix is 6 integer, 4 FP, 1 branch, 2 store, and 2 loads per cycle. The load scheduler is address-less and uses a 1K-entry modified Store Sets predictor. The scheduler models selective replay [8] for instructions dependent on loads that miss in the cache or that forward from an SQ whose access latency is longer than cache latency. The primary caches are 64KB, 2-way set-associative, and 3-cycle access. The L2 is 1MB, 8-way set-associative, and 10 cycle access. The TLBs are 128-entry, 4-way set-associative. Memory latency is 150 cycles. The L2 and memory buses are 16B wide, the latter is clocked at 1/4 processor frequency.

**SQ relevant structures.** Our processor uses SVW-filtered re-execution to verify speculation associated with both memory ordering and forwarding (our baseline processor uses it to verify only memory ordering). The SVW mechanism uses 16-bit SSNs, a 2K-entry 1-byte granularity SSBF with 2 read and 2 write ports, and a similarly configured SPCT. The FSP and DDP are 4K-entry 2-way set-associative with 2 read and 2 write ports. ***Indexed forwarding requires a larger FSP*** (4K-entry rather than 1K) because it requires all in-flight store-load dependences, not only ones that execute out-of-order. The SAT has 256 entries. The SAT has 6 read ports (2 for each of two loads renamed per cycle, 1 for each of 2 stores renamed per cycle to allow logging for SAT repair) and 2 write ports (1 for each of 2 stores renamed per cycle). It supports 4 checkpoints.

The sizes of these structures can be calculated from the SSN width (2B), the SQ size, and the SAT size (256-entries). The SSBF and SAT hold SSNs, so their capacities are 4KB and 512B, respectively. Each DDP entry holds two delay distances (each bounded by SQ size) and a 4-bit counter, for a total of 2B. Assuming 1B tags, a 4K-entry DDP represents 12KB of storage. Because the SAT is untagged and is indexed using only 8-bits (1B), the FSP and SPCT may represent store PCs using only 1B. Assuming 1B tags and 4-bit counters for the FSP, these would be 10KB and 2KB, respectively.

### 4.2. Quantitative Store Queue Comparison

To quantify the scalability differences between associative and indexed SQ designs, we use CACTI 3.2 [18]—modified to simulate memories of arbitrary configurations—to calculate the load latencies and energies of SQs with different capacities and load bandwidths. For all calculations, we use 90nm technology, a 1.1V supply voltage, and a 3GHz clock. Although the absolute numbers may not be accurate, we expect the trends to be representative.

**Configuration.** To avoid aliasing, SQs hold physical addresses. To sidestep the latency of address translation in SQ access, modern designs use the analog of a virtually-indexed/physically-tagged cache and access the SQ CAM only with the untranslated low-order address bits (i.e., the page offset). The remaining physical address bits are recorded in the SQ RAM and are used to perform a cache-style full address match on the selected entry after TLB access. This approach has the side benefit of reducing CAM width and latency. We assume that 64-bit data, 40-bit physical addresses, and 4KB pages. For the associative SQ the partial-address CAM is 12 bits wide and the RAM is 96 bits wide (64 data + 28 remaining address + 4 size/ready). Indexed SQ RAM entries are 108 bits wide; there is no indexed SQ CAM.

**Latency.** Table 2 shows the load latencies for SQs, data cache banks, and a TLB. All SQs have one indexed write port for store execution and one indexed read port for store commit. An associative SQ with two load ports (for an 8-way issue, 512-entry re-order buffer processor) has a load latency of 1.38ns (5 cycles at 3GHz); this estimate does not include the age logic. A comparable indexed SQ has a latency of only 0.60ns (2 cycles). Indexed SQ latency can be reduced by banking; the age logic makes banking an associative SQ more difficult.

The most significant aspect of SQ latency is its relationship to data cache latency. Our latency estimate for a 2-way interleaved 64KB data cache (i.e., for a single 32KB bank) is 1.00ns (3 cycles). To maximize performance, processors speculatively schedule load-dependent instructions assuming data cache access latency for the load. If SQ latency is equal to or less than cache latency—as for the indexed SQ—the scheduler can assume data cache latency for SQ-forwarded loads (which are the minority), effectively ignoring the for-

| | | 1 Load Port | | 2 Load Ports | |
|---|---|---|---|---|---|
| | | Assoc. | Index | Assoc. | Index |
| SQ | **16-entry** | 0.98 (3) | 0.51 (2) | 1.01 (3) | 0.53 (2) |
| | 32-entry | 1.12 (4) | 0.53 (2) | 1.14 (4) | 0.55 (2) |
| | **64-entry** | 1.34 (4) | 0.57 (2) | **1.38 (5)** | **0.60 (2)** |
| | 128-entry | 1.51 (5) | 0.67 (2) | 1.55 (5) | 0.71 (3) |
| | 256-entry | 1.73 (6) | 0.70 (3) | 1.79 (6) | 0.75 (3) |
| D$ bank | 8KB, 2-way | 0.84 (3) | | 0.92 (3) | |
| | 32KB, 2-way | **1.00 (3)** | | 1.15 (4) | |
| TLB | 32-entry, 4-way | 0.64 (2) | | 0.70 (3) | |

**Table 2. Store queue latencies in 90nm process.** ns and equivalent cycles on a 3GHz processor.

ward/no-forward distinction. If SQ latency is longer than cache latency—as for a large associative SQ—the scheduler has several options. First, it could treat all loads as having SQ latency. Because loads that do not forward dominate, this approach is not attractive. Alternatively, it could speculatively treat all loads as having cache latency, then handle forwarding like a cache miss and replay dependent instructions. This approach incurs expensive replays and suffers as windows grow and forwarding becomes more prevalent. Finally, the scheduler could hybridize these two approaches and predict (e.g., using the store-load pair predictor) whether a given load will forward. This form of "forwarding prediction" was implicitly used in a segmented SQ [13], we believe that our use of it in the context of a conventional SQ is novel. We model both the second and third approaches.

**Energy.** Although not shown in the table, our energy calculations show that for 64 entries and 2 load ports, the per-access energy of an indexed SQ is about 30% lower than that of an associative SQ. The difference is this "low" because the energy-hungry CAM is only 12-bits wide. Regardless, a 30% advantage combined with more natural support for energy-saving organizations like interleaving suggests the potential for significant SQ energy savings. However, our experiments indicate that associative SQ energy accounts for an average of 1.5% of total processor energy for our configuration. So although converting the SQ from associative to indexed saves SQ energy, total energy consumption largely mirrors execution time. For this reason, we do not further quantify the energy impact of our technique.

## 4.3. Forwarding and Delay Prediction

Accurate SQ index prediction is at the heart of our proposed design. To avoid introducing performance degradation, the mis-forwarding rate must be low.

The first (shaded) column of Table 3 shows the load forwarding rate (percentage of dynamic loads that forward). Across all benchmark suites, the load forwarding rate is 12.9%, although individual benchmarks (*vortex*, *mesa*, *sixtrack*, *gs*, *mpeg2*) forward at much higher rates. The complement of the forwarding rate (i.e., the percentage of loads that rightfully get their values from the cache) is the lower-bound accuracy for our forwarding index predictor. Because we match addresses prior to

| | | Fwd | Fwd+Dly | | |
|---|---|---|---|---|---|
| | %load forward | mis-forward /1000 | mis-forward /1000 | %load delay | avg. delay cycles |
| adpcm.d | 0.0 | 0.0 | 0.0 | 0.0 | 7.6 |
| adpcm.e | 0.0 | 0.0 | 0.0 | 0.0 | 6.8 |
| epic.e | 8.6 | 0.3 | 0.2 | 0.1 | 31.5 |
| epic.d | 19.2 | 0.1 | 0.1 | 0.2 | 11.0 |
| g721.d | 7.4 | 0.0 | 0.0 | 0.4 | 15.7 |
| g721.e | 10.5 | 1.7 | 0.0 | 0.3 | 6.4 |
| gs.d | 26.5 | 3.0 | 0.1 | 6.5 | 28.9 |
| gsm.d | 3.0 | 1.4 | 0.4 | 2.9 | 9.8 |
| gsm.e | 7.2 | 2.2 | 0.1 | 3.8 | 23.0 |
| jpeg.d | 1.7 | 0.3 | 0.4 | 2.0 | 35.5 |
| jpeg.e | 14.3 | 1.2 | 1.2 | 0.3 | 22.2 |
| mesa.m | 43.6 | 1.9 | 0.0 | 0.6 | 30.0 |
| mesa.o | 39.2 | 0.2 | 0.2 | 0.1 | 25.0 |
| mesa.t | 35.9 | 12.3 | 0.8 | 5.3 | 72.6 |
| mpeg2.d | 25.2 | 0.3 | 0.0 | 0.2 | 16.7 |
| mpeg2.e | 4.8 | 0.2 | 0.2 | 0.1 | 31.8 |
| pegwit.d | 8.4 | 2.0 | 0.4 | 1.6 | 19.5 |
| pegwit.e | 9.2 | 3.7 | 0.5 | 1.3 | 29.3 |
| **Media.avg** | **14.3** | **1.6** | **0.1** | **2.1** | **32.5** |
| bzip2 | 11.7 | 1.9 | 0.4 | 1.3 | 36.9 |
| crafty | 7.0 | 1.2 | 0.3 | 1.1 | 31.3 |
| eon.c | 28.4 | 5.0 | 0.8 | 8.3 | 21.0 |
| eon.k | 21.0 | 7.0 | 0.9 | 8.0 | 19.7 |
| eon.r | 24.2 | 7.1 | 0.9 | 9.5 | 23.3 |
| gap | 9.5 | 0.5 | 0.1 | 0.5 | 41.2 |
| gcc | 9.2 | 0.9 | 0.2 | 2.2 | 21.0 |
| gzip | 19.6 | 1.2 | 0.2 | 1.6 | 32.4 |
| mcf | 2.6 | 1.3 | 0.4 | 1.1 | 95.3 |
| parser | 14.0 | 4.3 | 0.2 | 1.8 | 65.8 |
| perl.d | 10.8 | 0.9 | 0.1 | 0.9 | 15.9 |
| perl.s | 12.7 | 0.9 | 0.0 | 0.3 | 11.2 |
| twolf | 9.7 | 2.9 | 1.0 | 1.2 | 18.5 |
| vortex | 24.5 | 3.7 | 0.2 | 2.8 | 29.4 |
| vpr.p | 8.4 | 1.9 | 0.5 | 1.2 | 15.6 |
| vpr.r | 18.9 | 0.9 | 0.4 | 0.6 | 67.7 |
| **Int.avg** | **13.5** | **1.8** | **0.3** | **1.6** | **53.2** |
| ammp | 13.7 | 3.3 | 0.2 | 1.0 | 90.4 |
| applu | 13.1 | 1.6 | 0.0 | 0.4 | 43.5 |
| apsi | 6.9 | 0.7 | 0.5 | 2.2 | 237.6 |
| art | 2.0 | 0.0 | 0.0 | 0.9 | 406.4 |
| equake | 4.2 | 0.6 | 0.4 | 0.8 | 75.5 |
| facerec | 2.0 | 0.0 | 0.0 | 0.4 | 62.8 |
| galgel | 1.7 | 0.8 | 0.1 | 0.3 | 51.4 |
| lucas | 0.0 | 0.0 | 0.0 | 0.2 | 34.0 |
| mesa | 25.4 | 3.3 | 0.1 | 5.9 | 92.4 |
| mgrid | 5.5 | 1.1 | 0.0 | 0.5 | 19.4 |
| sixtrack | 33.9 | 9.5 | 2.4 | 8.8 | 38.2 |
| swim | 3.2 | 0.1 | 0.0 | 0.4 | 105.4 |
| wupwise | 18.4 | 2.5 | 0.9 | 11.8 | 52.9 |
| **FP.avg** | **11.5** | **1.9** | **0.3** | **3.2** | **100.0** |
| **All.avg** | **12.9** | **1.8** | **0.3** | **2.3** | **53.1** |

**Table 3. Store queue index prediction diagnostics.** Load forwarding rates, raw prediction accuracy, and improved accuracy using delay prediction.

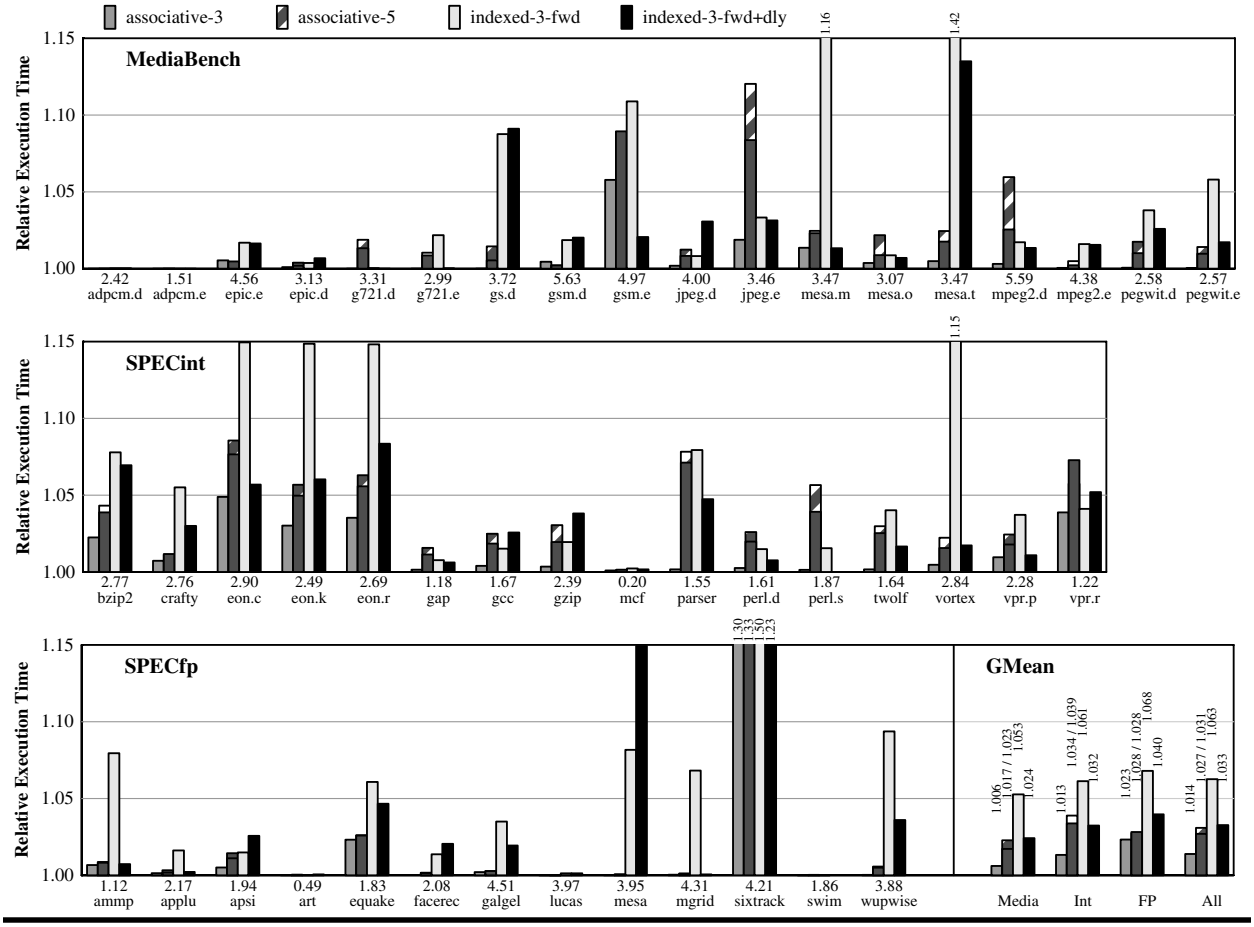


**Figure 4. Performance.** Execution times relative to an ideal, 3-cycle associative store queue with oracle load scheduling.

forwarding, we cannot possibly mis-forward these loads. Fortunately, our predictor is much more accurate than this lower bound.

Table 3 also shows dynamic load mis-forwardings per 1000 loads. The *Fwd* configuration represents raw accuracy with no delay prediction. The *Fwd+Dly* configuration adds delay prediction, and also lists the percentage of loads delayed and the average number of delay cycles per delayed load. Without delays, our forwarding predictor only mis-forwards on average 1.8 times per 1000 loads. Put another way, it induces pipeline flushes less frequently than control speculation driven by a typical branch predictor. Adding delay prediction reduces the mis-forwarding rate to 0.3 loads per 1000 at the cost of on average delaying about 2% of dynamic loads. More importantly, our delay predictor substantially reduces flushing for benchmarks with high mis-forwarding rates (e.g., *eon*, *sixtrack*, and *mesa.texgen*). For example, for *mesa.texgen* mis-forwarding drops from 12.3 to 0.8 per 1000 loads.

## 4.4. Performance

Figure 4 shows execution times of five different SQ configurations relative to an ideal baseline: a 64-entry associative SQ with 3-cycle access (same as data cache) and oracle scheduling. The IPC of this idealized configuration is printed above the benchmark name. Because we use relative execution times, shorter bars are better (as they represent lower overhead versus our idealized baseline). When reporting average relative performance, we use the geometric mean.

***Associative-3*** (first bar from left) is an associative SQ with ideal 3-cycle latency, and our formulation of Store Sets scheduling. Load scheduling overheads cause only a 1.4% slowdown over idealized load scheduling. The overhead is less than 1% for most benchmarks, and only *sixtrack* and *gsm.e* have more than a 5% overhead. Both suffer from one of the limitations of our particular Store Sets formulation: the inability to represent more than 2 (FSP associativity) store dependences per load. However, our experiments show that in many other cases our formulation slightly outperforms the original.

***Associative-5*** (second bar) is an associative SQ with a 5-cycle access latency and (modified) Store Sets scheduling. Two sub-configurations are shown as a stack. In the first (striped, top portion of the stack), the scheduler optimistically assumes a 3-cycle load latency; forwarding triggers dependent instruction replays. In the second, the scheduler uses Store Sets to predict which loads will forward and avoid some dependent-instruc-

tion replays. The performance penalty (over a 3-cycle SQ with realistic scheduling) of a 5-cycle design is 1.7% on average. Forwarding prediction reduces this penalty to about 1.3%, but it actually decreases performance for programs with individual loads that forward with low but non-zero frequencies (e.g., *vpr.route*). In the ensuing discussion, we compare to the 5-cycle associative SQ that exploits forwarding prediction.

***Indexed-3-fwd*** is our new 3-cycle indexed SQ without delay prediction. Even without delay, indexed forwarding incurs an average slowdown of only 5% relative to a 3-cycle SQ with realistic scheduling, and it is only 3.6% slower than a 5-cycle associative SQ. However, without delay prediction, some programs with significant rates of not-most-recent forwarding (e.g., *mesa.texgen*, *bzip2*, *ammp*, *equake*, and *wupwise)* exhibit large slowdowns.

Delay prediction helps address the not-most-recent forwarding pathology. ***Indexed-3-fwd+dly*** is our indexed SQ with delay prediction. With delay, indexed forwarding is 3.3% slower than an idealized 3-cycle SQ and only 0.6% slower than the 5-cycle associative SQ. The performance advantage of the 5-cycle associative SQ is concentrated in 16 benchmarks. The indexed SQ outperforms the associative SQ on 19 of 47 benchmarks. On 12 others—generally programs with little forwarding—the two have similar performance. Despite the addition of delay, the 5-cycle associative SQ remains superior for programs with high not-most-recent forwarding rates (e.g., *bzip2*, *mesa.texgen, equake, wupwise*). This is not surprising because delay prediction does not completely eliminate the performance penalty of not-most-recent forwarding; it simply converts the flushing penalty to a less severe delay penalty. This is sufficient to narrow the performance gap, often substantially. It is typically not sufficient to overcome the natural advantage of associative search, which can actually perform not-most-recent forwarding.

In addition to reducing performance overhead for not-most-recent forwarding, delay prediction also helps with FSP conflict misses. This is the effect in *eon* and *vortex*. Without delay, loads that forward from a large number of static stores thrash in the FSP and flush frequently. With delay, these loads still thrash, but they are also delayed long enough to avoid flushing.

Interestingly, delay prediction actually *degrades* the performance of 6 of the 47 benchmarks. Programs like *jpeg.decode*, *gcc*, *gzip*, and *mesa* prefer to forward aggressively with no delay. This result helps to put delay prediction in proper context. Delay is not a universally beneficial mechanism. It targets and suppresses the performance loss associated with certain indexed access pathologies like not-most-recent forwarding and FSP conflicts. In the process, it introduces delays into well-behaved indexed forwarding. On average, the beneficial outweighs the harmful, yielding an overall performance improvement.

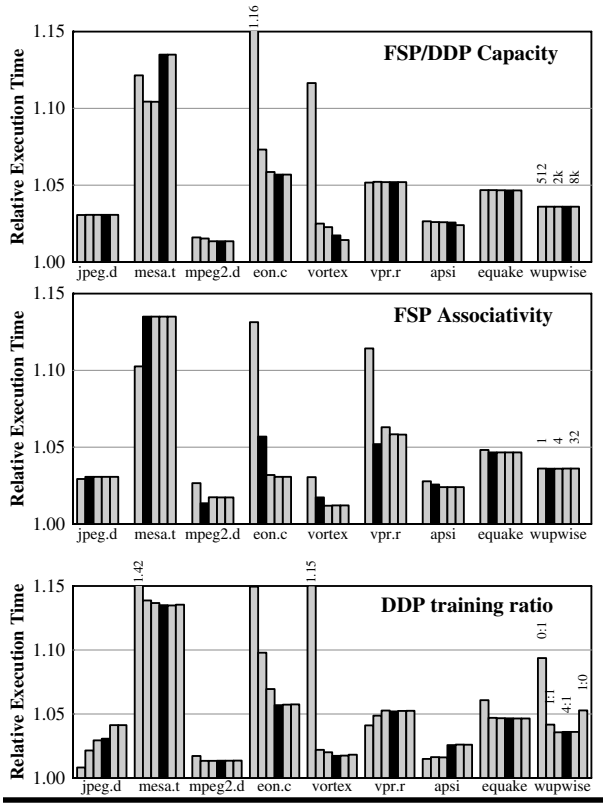


**Figure 5. Performance sensitivity.** Normalized runtime for the store queue with different forwarding and delay prediction configurations. In all graphs, the black bar is our default configuration.

## 4.5. Performance Sensitivity

Any prediction-based scheme has a wide range of possible predictor configurations. In this subsection, we explore the performance sensitivity of our proposed SQ to three predictor design dimensions. We perform this analysis using three benchmarks from each suite.

**FSP/DDP capacity.** The top graph of Figure 5 shows the effect of varying the capacities of 2-way set-associative FSPs and DDPs (in conjunction), from 512 to 8K entries by factors of two. Our default 4K-entry configuration is in black. As expected, smaller tables trade some performance in exchange for reduced implementation cost. Even a 1K-entry FSP often performs as well as a much larger table; our default 4K-entry FSP is actually over-provisioned for most programs. Performance begins to degrade at 512 FSP entries, especially for programs with large static load-store dependence footprints (unlike scheduling, indexed forwarding requires *all* in-flight store-load dependences to be represented). The *mesa.texgen* result displays an anomaly in our mechanism. Up to 2K-entries, the benefits of increased FSP capacity dominate. However, after 2K-entries increased DDP capacity leads to over-delaying.

**FSP associativity.** The middle graph in Figure 5 shows the effects of varying associativity for a 4K-entry FSP; DDP associativity is fixed at 2. The bars correspond to associativities of 1, our default 2, 4, 8, and 32.

Reducing associativity to 1 increases overheads dramatically. Many benchmarks have at least a few loads that forward from more than one static store. In contrast, few benchmarks benefit from higher associativities.

Note, although we vary FSP associativity, we maintain the invariant of accessing the SQ at most once per load. Breaking this invariant has the potential to overcome our not-most-recent forwarding shortcoming, but will also complicate our design and partially negate the transition from associative search to indexed lookup.

**DDP training ratio.** The final graph in Figure 5 measures sensitivity to the DDP training ratio. In the first bar from the left, delay is trained with a positive:negative ratio of 0:1; in other words it is never trained and effectively degenerates to the "raw" *Fwd* configuration. Successive ratios are 1:1 (delay and no delay are equally weighted), 2:1, our default 4:1, 8:1 and 1:0 (delay is never "un-learned"). Although many benchmarks are insensitive to the DDP training ratio, some benchmarks (e.g, *jpeg.decode*) prefer lower ratios (i.e., to flush rather than delay) while others (e.g., *eon.c*) prefer high ratios (i.e., to delay rather than flush). For most benchmarks, our default 4:1 ratio provides a good compromise between over- and under- delay.

## 5. Related Work

We have already discussed work related to load scheduling [3, 7, 9, 12, 22] and filtered re-execution [2, 6, 16] in our background section and in the exposition of our technique. In this section we focus on competing designs for scalable SQs.

**Age-ordered SQs.** One class of designs maintains the age-ordered SQ structure but uses partitioning, filtering, and hierarchy to improve its bandwidth and capacity scalability. Sethumadhavan et al. [17] scale SQ access bandwidth by guarding the SQ with a Bloom filter that conservatively encodes the addresses of in-flight stores. Only loads whose addresses hit in this filter access the SQ. This scheme is generally effective, but suffers from several drawbacks. Specifically, the Bloom filter is managed speculatively and out-of-order meaning that its contents are difficult to maintain precisely and that it is vulnerable to false positives from loads that match younger (i.e., non forwarding) stores. It also adds to the load execution critical path.

Srinivasan et al. [19] apply a similar strategy to a two-level SQ. A fast first-level SQ holds the most recent stores while a larger, second-level SQ holds all in-flight stores. A Bloom Filter eliminates most searches to the second-level SQ. A more recent version of their design eliminates the associative function of the second-level SQ by allowing old stores to speculatively spill to the data cache and implementing speculative forwarding through the cache [5].

Park et al. [13] scale SQ access bandwidth using a store-load dependence predictor (like ours) modified to track all in-flight dependences. They scale SQ capacity (and bandwidth) by chaining multiple SQ segments together and accessing them in a pipelined fashion. The disadvantage of this scheme is that it introduces variability in load latency, complicating the scheduler.

Roth [15] and Baugh and Zilles [1] propose to scale SQ size and bandwidth by dividing the store-commit/store-load forwarding functions of a conventional SQ between two queues. A large commit SQ contains all stores but is not associatively searched. Forwarding is implemented using a small associative SQ that contains only stores whose previous instances forwarded to loads and is accessed only by loads whose previous instances required forwarding. Re-execution or some other form of high-bandwidth verification detects loads and stores that were falsely excluded from the forwarding SQ and trains the store-membership/load-access predictor. This design generally performs well, but suffers on programs where a large fraction of dynamic stores forward to future loads. For these, a small set-associative "best-effort" forwarding structure can off-load some of the forwarding from the associative SQ.

These techniques improve SQ scalability but maintain the basic associative search functionality. Our design is the first of which we are aware that completely eliminates associative search in an age-ordered SQ.

**Address-indexed SQs.** A more inherently scalable alternative to an age-ordered SQ design is an address-indexed design as proposed by Torres et al. [20]. In addition to replacing fully-associative search with set-associative search, an address-indexed SQ supports interleaving that matches that of the data cache. However, there are many disadvantages to an address-indexed SQ [17]: it suffers from address conflicts, its contents are difficult to maintain precisely in the presence of control and data mis-speculations, and it does not naturally support multiple in-flight versions of the same address (although these can be supported using explicit age tags). To be effective, address-indexed forwarding must be treated as speculative and backed by a conventional forwarding mechanism [15, 20].

Multiscalar's ARB [4] is an address-indexed forwarding and disambiguation structure. It does not suffer from some of the traditional limitations of address-indexed designs because it does not track all in-flight stores, but rather only the "live-out" stores of each of a fixed number of processing elements (PEs). In an ARB, the number of in flight versions of each address is limited to the number of PEs; so these can be explicitly tracked. ARB stores are also non-speculative with respect to their PE, making ARB recovery an infrequent event amenable to a simple low performance implementation.

**Speculative memory renaming.** The FSP and SAT essentially implement speculative memory renaming. Just like a register alias table (RAT, register rename map) directly connects register consumers to their in-flight producers, the SAT directly connects memory consumers (loads) to their in-flight producers (stores). But where a RAT makes these connections non-speculatively using register names, the SAT makes them specu-

latively using the store-load dependence information in the FSP. Previous implementations of memory renaming [10, 14, 21] focused on reducing execution latency for a subset of forwarding loads by collapsing DEF-store-load-USE chains to DEF-USE chains, conventional associative forwarding is used for the rest. We use memory renaming not to collapse dependence-chains, but rather to eliminate associative search for *all* loads.

## 6. Conclusions

Traditional, associatively-searched structures for tracking in-flight memory operations are a timing bottleneck for future large-window processor designs. An associative store queue (SQ) for a processor with a 512-entry instruction window may have significantly higher latency than the first-level data cache, reducing performance and increasing the complexity of a deeply pipelined design.

This work introduces an SQ design that implements store-load forwarding without associative search. For each load the processor predicts a single SQ entry to query, transforming SQ lookup into a cache-style direct-indexed lookup. To support indexed access, we introduce two predictors. A PC-based forwarding predictor inspired Store Sets [3] identifies likely forwarding SQ entries. A distance based delay predictor inspired by the Exclusive Collision Predictor [22] delays difficult-to-predict loads until the stores they are likely to forward from have committed.

Detailed timing simulations of the SPEC2000 and MediaBench benchmarks shows that this design yields a 3.3% average slowdown relative to an idealized associative SQ (same access latency as data cache) and only a 0.6% slowdown relative to realistic associative SQ (longer access latency than data cache). The indexed SQ beats the realistic associative SQ on 19 of 47 programs and matches it on 12 others. The indexed SQ also has non-performance implementation advantages. It unifies (and simplifies) load-scheduling and store-load forwarding and avoids forwarding-related instruction replays. The combination of our indexed SQ with a previous design that uses filtered load re-execution to eliminate load queue search [2] yields a more scalable system for managing in-flight memory operations: one that uses no associative search whatsoever.

Although our predictor is reasonably accurate, it does have several limitations, most notably an inability to forward from not-most-recent store instances. Future work should explore alternative predictor organizations and approaches, potentially with an aim to overcome these limitations. For example, path-based information might increase both forwarding prediction and delay prediction accuracy and robustness.

## Acknowledgments

We thank the anonymous reviewers for their comments and suggestions. This work was partially supported by NSF CAREER Award CCF-0238203 (Roth).

## References

[1] L. Baugh and C. Zilles. "Decomposing the Load-Store Queue by Function for Power Reduction and Scalability." In *2004 IBM P=AC^2 Conference*, Oct. 2004.

[2] H. Cain and M. Lipasti. "Memory Ordering: A Value Based Definition." In *Proc. 31st International Symposium on Computer Architecture*, pages 90–101, Jun. 2004.

[3] G. Chrysos and J. Emer. "Memory Dependence Prediction using Store Sets." In *Proc. 25th International Symposium on Computer Architecture*, pages 142–153, Jun. 1998.

[4] M. Franklin and G. Sohi. "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References." *IEEE Transactions on Computers*, May 1996.

[5] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasan, and K. Lai. "Scalable Load and Store Processing in Latency Tolerant Processors." In *Proc. 32nd International Symposium on Computer Architecture*, pages 446–457, Jun. 2005.

[6] K. Gharachorloo, A. Gupta, and J. Hennessy. "Two Techniques to Enhance the Performance of Memory Consistency Models." In *Proc. of the International Conference on Parallel Processing*, pages 355–364, Aug. 1991.

[7] R. Kessler. "The Alpha 21264 Microprocessor." *IEEE Micro*, 19(2), Mar./Apr. 1999.

[8] I. Kim and M. Lipasti. "Understanding Scheduling Replay Schemes." In *Proc. 10th International Symposium on High Performance Computer Architecture*, Feb. 2004.

[9] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. "Dynamic Speculation and Synchronization of Data Dependences." In *Proc. 24th International Symposium on Computer Architecture*, pages 181–193, Jun. 1997.

[10] A. Moshovos and G. Sohi. "Streamlining Inter-Operation Communication via Data Dependence Prediction." In *Proc. 30th International Symposium on Microarchitecture*, pages 235–245, Dec. 1997.

[11] A. Moshovos and G. Sohi. "Memory Dependence Speculation Tradeoffs in Centralized, Continuous-Window Superscalar Processors." In *Proc. 6th Annual International Symposium on High-Performance Computer Architecture*, pages 301–312, Feb. 2000.

[12] S. Onder and R. Gupta. "Dynamic Memory Disambiguation in the Presence of Out-of-Order Store Issuing." In *Proc. 32nd International Symposium on Microarchitecture*, pages 170–176, Nov. 1999.

[13] I. Park, C. Ooi, and T. Vijaykumar. "Reducing Design Complexity of the Load/Store Queue." In *Proc. 36th International Symposium on Microarchitecture*, Dec. 2003.

[14] V. Petric, A. Bracy, and A. Roth. "Three Extensions to Register Integration." In *Proc. 35th International Symposium on Microarchitecture*, Nov. 2002.

[15] A. Roth. "A High Bandwidth Low Latency Load/Store Unit for Single- and Multi- Threaded Processors." Technical Report MS-CIS-04-09, University of Pennsylvania, Jun. 2004.

[16] A. Roth. "Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization." In *Proc. 32nd International Symposium on Computer Architecture*, pages 458–468, Jun. 2005.

[17] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. "Scalable Hardware Memory Disambiguation for High ILP Processors." In *Proc. 36th International Symposium on Microarchitecture*, Dec. 2003.

[18] P. Shivakumar and N. Jouppi. "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model." Technical report, COMPAQ Western Research Laboratory, 2001.

[19] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. "Continual Flow Pipelines." In *Proc. 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[20] E. Torres, P. Ibanez, V. Vinals, and J. Llaberia. "Store Buffer Design in First-Level Multibanked Data Caches." In *Proc. 32nd International Symposium on Computer Architecture*, pages 469–480, Jun. 2005.

[21] G. Tyson and T. Austin. "Improving the Accuracy and Performance of Memory Communication Through Renaming." In *Proc. 30th International Symposium on Microarchitecture*, pages 218–227, Dec. 1997.

[22] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. "Speculation Techniques for Improving Load-Related Instruction Scheduling." In *Proc. 26th Annual International Symposium on Computer Architecture*, pages 42–53, May 1999.